P. Findeisen, P. Gburzynski

G A S S

MERA-400 ASSEMBLY LANGUAGE

R E F E R E N C E M A N U A L

VERSION 6.0

PART ONE

IIUW, WARSZAWA, 1981

REVISED IN NOVEMBER 1987

Introduction

Contained herein is a reference manual of the assembly language Gass which, apart from other features, provides a symbolic notation for the Mera-400 processor instruction set. Gass has been designed and implemented at IIUW in 1981 as a part of a larger project aiming at the implementation of an operating system for the computer in question.

The manual consists of two self-contained parts. Part one presents the (almost) machine-independent features of Gass. Part two describes the machine architecture, instruction formats and opcodes.

The limited size of this manual made it impossible to include an introduction to programming directed at a more general audience: the reader is expected to have had some experience with a reasonable assembler.

The authors wish to express their kind thanks to all those who by any means contributed to the present version of Gass and this manual. In particular we are profoundly indebted to P. Raszkowski who originated the concept of the instruction mnemonics and for a long time actively participated in the entire project. The considerable help of P. Carlson (who suggested many details) is also highly appreciated. This manual was carefully read by a number of our friends including J. Deminet, W. Dobosiewicz and F. Kluzniak. Their critical and constructive notes helped us to avoid many mistakes, errors and horrors.

Caution

The user should check her version of Gass against inconsistency with the version number of this manual. In the case of different versions the information contained in the manual should be viewed with **limited faith**.

Contents

i. Input line format 5

i.1 Line fields 5
i.2 Tabulation 6
i.3 Comments 6
i.4 Symbolic names 8

ii. Blocks 10

iii. Symbolic values and their attributes 11

iii.1 Relocation types 11
iii.2 External variables and imported values 12
iii.3 Compound symbols 14

iv. Constants 15

iv.1 Numeric constants 15
iv.2 Long constants 17

v. Expressions 19

v.1 Symbolic arithmetic 19
v.2 Evaluable and non-evaluable expressions 19
v.3 Expression syntax 22
v.4 Address and numeric expressions 22

vi. Assigning values to symbols 25

vi.1 .equ - explicit assignment to a symbol 25
vi.2 Redefinability of symbols 27
vi.3 Changing the location counter 27

vii. Reservation pseudos 28

vii.1 .data 28
vii.2 .text 29
vii.3 .res 30

viii. Specific pseudos of the byte block 31

viii.1 .byte 31
viii.2 .resb 32

ix. Machine instruction format 33

ix.1 Register designators 33
ix.2 First argument 33
ix.3 Second argument 34

x. Macroassembly 39

x.1 Defining macros 39
x.2 Formal parameters 40
x.3 Actual parameters 40
x.4 Conditional macro expansion 41

xi. Micros 45

xii. Parametrized assembly 47

xiii. External assembly 48

xiv. Listing 49

xiv.1 Listing control pseudos 49

xv. Auxiliary pseudos 51

Appendix A 52

Appendix B 53

i. Input text format

Separate program modules that do not mutually interfere during the assembly time are called **"units"**. The input for Gass can consist of a number of units (in particular, one unit).

All the objects declared and/or referenced in a unit are identified by their **symbolic names**. The class of all such objects can be divided into the following subclasses:

i. Predeclared objects which are always well known to Gass. The user is not bound to declare, define or say anything about them. This subclass covers **pseudoinstructions** (Gass directives), **register descriptors** and the **machine instruction set**.

ii. User defined objects which are symbolic values (**addresses**, **numeric constants**) and **macrodefinitions**. This class is assumed empty whenever Gass starts to assemble a new unit.

i.1 Line fields

The source text to be assembled by Gass is usually supplied as the contents of a text file. Each line of that file is examined in order to decode its three main components which are:

location field
operation field
arguments

It may happen that any one, two or even three of these goodies are absent. In such a case we say that the corresponding field of the input line is empty.

The location field usually contains a symbolic name to which the current value of the **location counter** is to be assigned. Sometimes (e.g. .equ and .macro pseudos) the location symbol is treated in a special way.

The operation field contains the symbolic name of a **pseudo**, a **machine instruction mnemonic** or the name of a **macro**.

It is not hard to guess what is the purpose of the third component of the input line.

In a typical case (all exceptions will be mentioned successively) the location field starts from the very beginning of a line. If the line starts with a blank Gass assumes that the location field is empty and it skips over all the subsequent blanks. A nonempty location field continues over until the first blank is encountered. Then any further blanks are skipped and Gass starts to decode the operation field. The operation field is assumed present unless it starts with a character that cannot directly appear within a symbolic name. The first blank or comma terminates the operation field and is skipped together with any following blanks. The expected shape and form of the arguments depend usually on the contents of the operation field. The important point is

that **blanks do not terminate arguments!** They cannot split any single item and are material within text constants but ignored elsewhere.

i.2 Tabulation

Two-character sequences beginning with ht - horizontal tab (09 hexadecimal) can be used to tabulate and format the source text without overloading it with blanks. The code of the character which follows ht specifies the line position to be assumed for the remaining part of the current line. Each tabulating sequence is converted internally to the appropriate number of blanks. As it would be rather difficult to insert a negative number of blanks, each such a number (and also zero) is changed to one. Thus, whatever its follower, a horizontal tab always stands for a delimiter.

i.3 Comments

Any input line that starts with an **asterisk** or a **semicolon** is an all-comment line. All its fields are assumed empty and the line is ignored except for the listing. An empty line (i.e. a line that does not include any characters but blanks) has the same property.

Any other input line (in particular with non-trivial fields) may contain a **trailer comment** which is an arbitrary sequence of characters starting with a **semicolon**. Of course a semicolon occurring within a text constant stands for itself and is not considered as a character that starts any comments.

Remarks

Input text lines should start with control character (usually LF), followed by up to 120 characters terminated by the CR character. Longer lines cause errors and their contents are ignored. The first character of any line is always ignored, regardless of its value.

The user should refrain from using the characters **"null"** (00) and **"eot"** (04). They may occasionally cause funny results which cannot be precisely predicted and described.

Before going further into details let us follow the short program below. There is no use trying to understand just now what is going on in there - this program should be viewed as an illustration of the above introductory information.

Example

```
            .unit     gcd
*
*   This is a sample subroutine to compute the gcd of two positive
* integers passed by the caller in r1 and r2. The value is returned
* in r4.
*
            .exp      gcd             ; entry point is exported
*
divisor     .res      1               ; reserve cell for divisor
*
gcd         .res      1               ; space for the return address
*
*   We assume that the caller uses the =callm= instruction to call
* this subroutine.
*
*   As you have certainly noticed the entry point name doesn't differ
* from the name of the entire unit. Nevertheless those names have
* nothing in common.
*
loop         cmpn     r1,r2           ; are both numbers equal?
             b.eq  r2             ; are both numbers equal?
             b.eq        equal        ; jump if yes
*
             b.lt        reverse      ; jump if first arg. smaller
             s.rg     r2,divisor      ; send divisor to memory
             set      r2,r1           ; and move first arg. to r2
*
divide       clr      r1              ; clear r1
             div         divisor      ; r1:=mod(r2,divisor)
             l.rg     r2,divisor      ; fetch second argument
             b.uc        loop         ; and loop
*
reverse      s.rg     r1,divisor      ; divide by first argument
             b.uc        divide
*
equal        set      r4,r1           ; move result to r4
             buc       (gcd)          ; return
*
            .end
```

The simple program presented on the previous page can already teach us several interesting things. In particular we can see how to begin a unit. This is done by means of the following pseudo:

.unit "unitname"

The name of a unit can be any symbolic name legal for Gass. Refer to the next section for a detailed description of symbolic names.

The last statement of a unit should be the **.end** pseudo optionally followed by a symbolic address which represents the transfer point, i.e. the address from where the execution is to be started. The transfer symbol can be any local address symbol exported in the current unit (see iii.2).

All the names of the standard pseudos start with a **period**. However, the user's macros can define "private" pseudos that do not conform to that convention.

i.4 Symbolic names

Symbolic name is a sequence of characters that identifies an object. The length of a symbolic name is limited only by the maximum size of the input line. Nevertheless, only the initial **eight characters** of a name are significant and two names that do not differ in them are assumed equal. The following characters can be put directly into a symbolic name:

- **lower and uppercase letters**
- **non-initial digits**
- **the characters: $. : "underline"**

The lower and uppercase letters are assumed different as long as they are used to identify the user's private goodies. The symbolic names of pseudos and the instruction mnemonics are always assumed to be written in the lower case letters, i.e. all the upper case letters are converted by Gass to the lower case.

A symbolic name may also contain other characters (not included in the above class) but, if so, each of them must be explicitly preceded by the special escape character - **"\" (backslash)**. We present below a few examples.

Examples

legal symbolic names

```
a
.
$123
this\ is\ a\ good\ sample\ identifier
\1a
\\\\\\\\
\=\+\,abc\=
\a\dummy\escape
```

illegal names

```
1a          - name cannot begin with a digit.
bad ident   - blank illegal.
a100%       - percent illegal
\\\\\       - odd number of backslashes
```

ii. Blocks

The binary image of a unit is logically divided into three continuous blocks. They are:

byte
code
data

The programmer may assign her code to the specific block using the pseudo presented below.

.use "block identifier"

"Block identifier" may be any abbreviation of the block name, e.g. "d" and "dat" both identify the data block.

The location symbol is ignored if present. All the code assembled after an occurrence of the .use pseudo goes to the specified block until the next .use is encountered. Default block (assumed by Gass at the beginning of each new unit) is **"data"**.

The block structure has a number of advantages which facilitate the developement of the re-entrant programs. If the programmer observes some good habits then her pure code can easily be separated from data which is important to handle re-entrancy. The loader is responsible for the appropriate management of the blocks. In particular, it always attempts to fit the byte block into the lower 32k of the loadable memory area. Furthermore it guarantees that the zero address cell will not fall into the data or byte block. Thus the value zero may be commonly used as a nil pointer or sentinel for various data structures.

A separate **location counter** is provided for each block - it points to the next relative address in that block. Putting an item into a block affecve address in that block. Putting an item into a block affects its location counter which is incremented by the length of this item. For the byte block this length is expressed in bytes.

The value of the location counter for the block in use can be accessed via the special reserved symbol **"." (period)**. The relocation type (see iii.1) of this symbol always agrees with the type of the current block.

iii. Symbolic values and their attributes

The user-defined symbolic objects processed by Gass can be divided into two categories, namely **macros** and **symbolic values**. The latter category is the subject of the present section.

A symbolic value apart from the value itself possesses a number of attributes that specify what kind of value it is. Some of the attributes may change during assembly. For instance, a symbolic value may be **undefined** in any particular occurrence and **defined** in another one. Let us look at the following example:

Example

```
          .unit      sample
       ........................
sym1       lds    r3,1          ; sym1 becomes defined
       ........................
           b.uc       sym1        ; this is a reference to sym1
*
*   The above reference to sym1 is  a reference to a defined symbol.
* Below we refer to a symbol which is not yet defined. However, this
* reference is valid because the symbol is defined somewhere below.
*
       ........................
           i.bnz  r3,sym2         ; reference to sym2 - still undefined
       ........................
sym2       ptr        36          ; here sym2 becomes defined
          .end
```

Undefined symbolic names are not allowed in certain contexts. However, many references to undefined symbols are legal provided that the symbols are finally defined somehow.

iii.1 Relocation types

Another important property of a symbolic value is its relocation type. For each address symbol its relocation type is determined by the block in which it was defined. Values defined as offsets to other values inherit their relocation types. Non-address symbolic values, i.e. values which represent numbers rather than relocatable symbolic addresses, have null relocation types. We will say that they are **"absolute"** and **"numeric"**. Thus in fact we have four possible relocation types. It will be convenient to denote them by the following letters:

a - absolute value (null relocation type)
b - byte block relocation type
c - code block relocation type
d - data block relocation type

Let us get some more help from another example. Some of the gadgets

it brings out have not been described yet, and it is up to the reader's intuition to understand them now. If that intuition fails it would be advisable to look a few pages ahead where everything is presented and explained successively.

Example

```
            .unit      another\ sample
            .use       byte        ; assume byte block
label       .text      '\la few bytes to write\n'
*
*   =label= has been defined above as an address symbol in the byte
* block. Thus its relocation type is =b=.
*
            .use       code        ; assume code block
start1       l.byte r4,label
          ......................
*
*   The relocation type of =start1= is =c=.
*
          ......................
label1      .equ       label(5)
*
*   =label1= is an offset to =label=. Its relocation type is that of
* =label=, i.e. =b=
*
number      .equ       234 + 1981
cntr        .equ       2 * number + 1
*
*   Both the above symbols are assigned numeric absolute values. Thus
* their relocation type is =a=.
*
          ......................
            .end
```

iii.2 External variables and imported values

A symbolic value may be defined completely in a given unit or it may be fetched from outside. We say that a value is **local** in a unit if, given three relocation bases for the blocks of this unit, the value can be determined and computed. The values that are defined in other units or somewhere outside the current unit are called the **outer** values. They fit into two categories:

i. External variables

External variable is the name of an area that is to be shared by a number of units. The following pseudo can b to be shared by a number of units. The following pseudo can be used to declare an external variable:

"name" .xvar "numeric expression"

The location symbol must be present and will be used to identify the address of the shared area. The relocation type of an external variable

is borrowed from the current block, i.e. from the block being used when the corresponding .xvar pseudo has been encountered. The length of this variable is determined by the value of the obligatory argument. For b-type relocation this length is assumed to be in bytes and is rounded up to the full word boundary.

ii. Imported values

An imported value is a value that is referenced in one unit but defined (and local) in another. When the programmer wants to make use of an imported value she has to declare it as imported by means of one of the following pseudos:

```
.imp      "list of symbols"
.imc      "list of symbols"
```

"List of symbols" is a sequence of symbolic names separated by commas. Each symbolic name represents an imported value which can be an absolute constant (.imc pseudo) or an address (.imp pseudo). The relocation type of a symbolic address value imported by the .imp pseudo is determined by the type of the block currently used. The values imported by .imc are absolute numeric values so their relocation type is always "a".

Any symbolic value that is to be imported by any unit should be defined in some other unit and declared as an exported one. The following pseudo is used to export symbolic values from a unit:

```
.exp      "list of symbols"
```

All the symbols occurring in the argument list should be local in the unit they are exported from. In particular they cannot be imported from other units or declared as external variables. Furthermore, offsets to such symbols, as not considered local, are also inhibited in the export list.

If the name of a symbolic value declared as exported in one unit matches an external variable in another then both the exported value and the external variable are assumed to stand for the same object. One can make use of this fact when an external area has to be preloaded with data.

Example

```
         .unit      mainunit
komon    .xvar      200      ; declare external array of 200 words
         .use       bytes    ; assume byte block
exvar    .xvar      511      ; external array of 511 bytes
*
*   =komon= has been declared to represent the address of an external
* array of 200 elements. This array will be located somewhere in the
* data block as it was the block in use when komon has been declared.
*   =exvar= is an external array of 511 bytes. In fact this number
* will be rounded up to 512.
*
         .use       code     ; assume code block
         .imp       subr     ; declare imported (code) address =subr=
*
start     clr    r1           ; clear r1
          l.byte     r1,exvar(20)
       ......................
          call   r4,subr      ; jump to imported subroutine
       ......................
         .exp       start    ; don't forget to export trans. address
         .end       start    ; end unit, define transfer address

            .unit      secondunit
komon       .xvar      200          ; =komon= shared with previous unit
            .use       bytes        ; assume byte block
exvar       .byte      'a sample sequence of bytes'
         ......................
            .use       code
            .exp       subr         ; declare subr to be exported
subr         swab   r4
         ......................
            .exp       exvar
*
*   =exvar= is exported in the present unit but it has been declared
* as an external variable in the previous one. Since the relocation
* types agree in both occurrences of exvar everything is OK and both
* exvar's denote the same object. Thus the initial bytes of exvar
* referenced in mainunit are those defined in here.
*
            .end
```

iii.3 Compound symbols

Compound symbols are those defined as offsets to imported symbols or to external variables. Unfortunately, in a few sticky cases the usage of compound symbols is restricted, so they have to be named and classified.

iv. Constants

Constants are predefined objects which represent various kinds of values. They can be used to define symbolic values or to declare contents of the program memory area.

iv.1 Numeric constants

The constants that can appear in expressions and have 16-bit internal representations are called **numeric constants**. They are:

Integer constants

Integer constant is any sequence of digits (optionally preceded by a sign) which encode an integer value from the range **<-32768,32767>**.

Examples

```
                    1
                   -1
          00000000012345
                 -32768

            320911992992        - bad constant (value too large)
                   30a51        - bad constant (letter illegal)
                  -20 01        - bad constant (space illegal)
```

Hexadecimal constants

A hexadecimal constant starts from **"H"** or **"X"** followed by a sequence of hexadecimal digits enclosed in apostrophes. The hexadecimal digits are: 0,1,2,3,4,5,6,7,8,9,0,A,B,C,D,E,F where the decimals 0-9 stand for themselves and the letters A-F denote 10,11,12,13,14,15 respectively. The acceptable range of a hexadecimal constant is exactly four significant digits.

Examples

```
                    H'0'
                  X'FAAA'
           H'00000000001'
                     H''        - equivalent to H'0'

              H'11111111'       - illegal constant (too many digits)
                   H'10G'       - illegal constant (bad digit)
                  H'  0 '       - blanks are illegal
```

A hexadecimal constant, as well as all the numeric constants presented below, can be preceded by a sign. However, this sign is not included as a part of the constant itself but it is treated as an operator. Let us note that in the case of integer constants, the initial sign has to be treated in another way. The reason is obvious for everyone who understands that it is impossible to legitimately negate the minimal negative number in the two-complement binary notation.

Octal constants

Octal constant is a sequence of octal digits enclosed in apostrophes and preceded by the letter "O".

Examples

```
               O'7777'
                  O'2'
            O'177777'       - the maximal octal constant

               O'108'       - illegal constant (bad octal digit)
```

Binary constants

Binary constant is a sequence of zeros and ones announced by "B" and, as usual, enclosed in apostrophes.

Examples

```
          B'1010101010'
    B'1111111111111111'     - the maximal binary constant

             B'0000012'     - bad constant (2 is not a binary digit)
```

Decimal constants

Sometimes it is desirable to treat the leftmost bit of a 16-bit integer value as a significant binary position rather than a sign of a number in the two-complement notation. Decimal constant is an unsigned integer located between apostrophes and preceded by "D". This integer is treated as a non-negative number from **<0,65535>**.

Examples

```
                D'1'        - same as 1
            D'65535'        - equivalent of <-1>

            D'-3255'        - bad constant (sign illegal)
```

Character constants

Character constant consists of one or two characters enclosed in apostrophes. The value of this constant is determined by the Ascii code representations of the character(s) in question. If a character constant has to include an apostrophe then the apostrophe must be preceded by a backslash. The sequence **"\N"** (upper case "N") stands for the carriage return and is replaced by the **"cr"** (X'0d') character wherever it occurs. Similarly, **"\L"** denotes line feed (X'0a') character.

Examples

```
              'a'           - equivalent to X'41'
             'ab'           -     --//--    X'4142'
            '0\N'           -     --//--    X'300d'
```

```
      '''         - error ("'" not preceded by "\")
   'text'         - error (only two characters allowed)
```

In order to handle the R40-code constants one extra constant type has been introduced, it is:

C'"up to three code characters"'

Only letters and digits are allowed for code characters, and first character must be a letter.

Examples

```
C'IN'
C'A1B'
C'MON'

C'OK.'       - dot is illegal
```

iv.2 Long constants

Long constants represent the structures that cannot fit into one machine word. As a consequence, they cannot be used to define symbolic values - their exclusive purpose is to preload the binary image of a program with specific data.

Floating constants

The internal representation of a floating constant consists of three consecutive words (**48 bits**). The first **forty** bits of this structure represent the **two-complement mantissa** interpreted as a binary fraction from the range **<-1,1)**. The remaining **eight** bits are used to encode the binary **exponent** as a two-complement integer value. The resulting floating number would read as **mantissa*(2**exponent)**.

The non-zero mantissa of a floating constant must be left justified (**normalized**) so that its two leftmost bits be different. Thus in fact the proper range of a non-zero mantissa is always within one of the intervals <-1,-1/2), <1/2,1).

It is not hard to compute the bounds for the values of floating constants. From the expression above it follows that:

-1 * 2127 .ge. "floating constant" .lt. 1 * 2**127**

From the user's point of view a floating constant is a sequence of decimal digits preceded optionally by a sign and including either a period or the letter "E", or both the characters together. The period, as well as the letter "E", should be preceded by at least one decimal digit. Moreover "E" should be followed by at least one digit optionally preceded by a sign. The initial sequence of digits, which must be present, is called the **integer part** of the floating constant. The optional digits following the period constitute the **fractional part**. The obligatory (and possibly signed) follower of "E" is called **exponent**.

Examples

```
   number      internal representation in hexadecimal

     1.        X'4000'  X'0000'  X'0001'  = 2**1*(1/2)
    -1.0       X'8000'  X'0000'  X'0000'  = 2**0*( -1)
     0.0       X'0000'  X'0000'  X'0000'  = 2**0*( 0 )
     1E0       equivalent to 1. or 1.0    = 1 * 10**0
  1000.0E-10   X'6B5F'  X'CA6A'  X'F2E9'  = 1000*10**(-10)

     1         illegal floating constant (period or "E" required)
2000000.0E37   illegal floating constant (value too large)
      .123     illegal floating constant (should start with a digit)
```

Long integer constants

Long integer is a 32-bit two-complement number occupying two consecutive machine words. The value represented by a long integer must neither exceed **2147483647** nor be less than **-2147483648**. For the programmer, the long integers are their absolute decimal values written between apostrophes and announced by "L" optionally preceded by a sign.

Examples

```
              L'1'        - equivalent to  X'0000'  X'0001'
             -L'1'        -     --//--     X'FFFF'  X'FFFF'
      L'2147483647'       -     --//--     X'7FFF'  X'FFFF'
     -L'2147483648'       -     --//--     X'8000'  X'0000'

           L'123A0'       - error (illegal decimal digit)
          L'-11111'       - error (sign illegal here)
```

Text constant

The representation of a text constant has no fixed length. In fact, this representation depends slightly upon the context where the constant occurs. The purpose of a text constant is to cluster some characters to be printed out or to be processed as a text string. To write down a text constant, the programmer simply has to put down the desired string enclosing it in **apostrophes**. All the escaping properties of backslash with respect to the text constant are inherited from the character constant (see iv.1).

Regardless of the context, a text constant is never terminated by any special byte forced at its end. The user should ensure by her private means that her text constant terminates in the way she desires.

Examples

```
          'hey mr. tambourine man play a song for me'
        'sample backslash escape of \' (apostrophe) \n'

    'this is a sample erroneous text constant - ='= is illegal'
```

v. Expressions

Expression is an object which implicitly represents a 16-bit symbolic value together with a number of attributes. It may be composed of the following elements:

- **numeric constants**
- **symbolic values**
- **operators**
- **brackets: "<" ">"** ("less" and "greater")

The first two elements of this list have already been presented. Let us now describe the third one in detail.

v.1 Symbolic arithmetic

The following nine operators can appear in expressions:

1. **"+"** - arithmetic addition
2. **"-"** - subtraction
3. **"*"** - multiplication
4. **"/"** - division
5. **"!"** - logical "or"
6. **"&"** - logical "and"
7. **"%"** - exclusive "or"
8. **"@"** - scale (a @ b = a * 2**(15 - b))
9. **"#"** - a unary operator applicable only to the symbolic values and forcing the numeric type of these values.

The first four operators are the usual integer arithmetic performed on 16-bit two-complement operands. No overflow checks are performed except for division by zero and the result obtained may occasionally differ from the desired one.

The scale operator "@" is used to shift the value of the left argument to the bit position indicated by the value of the right argument. If it is negative or greater than 15 then its value modulo 16 is taken.

The unary operator "#" applied to a symbolic value clears its address attribute, i.e. the value is assumed to be a numeric one. However, the original relocation type is retained.

The operators 1 thru 8 do not have priorities and an expression is evaluated from left to right. However, in many cases brackets can be used to force preevaluaion for particular subexpressions.

v.2 Evaluable and non-evaluable expressions

The class of valid expressions can roughly be divided into two subclasses depending upon the difficulties in evaluating an expression.

Evaluable expressions

We say that an expression is evaluable if both its value and the relocation type can be determined entirely by Gass at the moment when those attributes are required.

Non-evaluable expressions

An expression is non-evaluable (still it may be valid) if Gass is not able to complete its evaluation. In such a case evaluation is left to the loader and it is performed (or merely finished) at load time.

Non-evaluable expressions are not permitted in certain contexts. If a non-evaluable expression occurs in such a context Gass flags an error and does not pass the evaluation to the loader.

Gass fails to complete the evaluation of an expression and assumes that it is facing a non-evaluable case if the step to be currently performed cannot produce one of the following results:

- **local absolute numeric value (a-type relocation)**
- **local relocatable value**
- **imported value +(-) absolute number**
- **address of an external variable +(-) absolute number**

It follows from the above definition that evaluable expressions can be characterized by the following properties:

i.
All the arguments of the operators 3 thru 8 are local absolute values (their relocation type is "a").

ii.
At least one argument of each addition is a local absolute value. The type of the remaining argument propagates then over the result.

iii.
For each subtraction either the second argument is local absolute (this reduces subtraction to an addition) or both the arguments are of the same kind, i.e. the result represents an absolute (and determinable) distance between them.

Any expression that does not conform to the above three rules is non-evaluable and, provided it is valid and permitted, its evaluation is left to the loader.

Before Gass acknowledges a legal non-evaluable case of an expression it carefully checks whether the loader will be able to perform the evaluation. The expression is definitely unevaluable if any of the following statements is true:

i.
A symbolic value occurring in the expression is never defined.

ii.
A subexpression enclosed in brackets cannot be reduced to a local absolute numeric value.

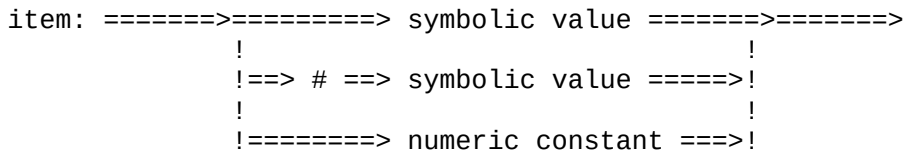
iii.
The tail that remains to be evaluated by the loader includes a compound symbol (see iii.3).

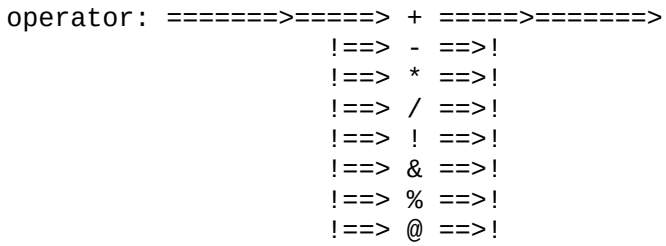
iv.
The second argument of a scale operator is not an absolute local value.

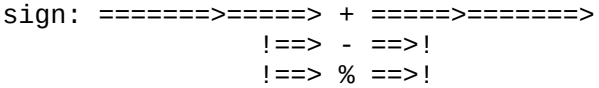
v.
The expression has a bad syntax.

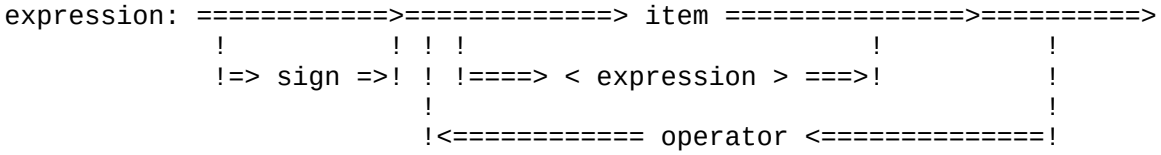
v.3 Expression syntax

Below we present the syntax diagram of an expression. However, we are not convinced that this picture offers the reader any reasonable help. The semantic notes contained elsewhere in the present section seem to be much more important.









An expression starting with a sign is virtually prefixed by a dummy operand. For the arithmetic signs that operand is obviously zero and for the logical sign "%" it is "all-ones", i.e. the logical complement of the successor is computed.

v.4 Address and numeric expressions

There is a number of situations where the assembler has to decide whether a given expression represents a value or an address of the value in question. In general, the problem is not a trivial one. Let us note that the virtual value of a non-evaluable expression cannot possess a relocation type. For instance, what would result if we divided a b-type symbolic address by a d-type imported value?

In order to solve this problem and to enable the programmer to force explicitly a specific type of an expression the following two notions have been introduced:

Address expression

An expression is an address expression (i.e. its value is assumed to

be an address) if it begins with an **unadorned address symbol** of any kind.

Numeric expression

Any expression which is not an address expression is a numeric one.

Let us note that without affecting values of expressions they may easily be converted at least in one direction. For instance, given an address symbol, say "dobo", we may consider two simplest expressions including this symbol, namely: "dobo" and "#dobo". The first one is obviously an address expression, the second - a numeric one. Thus the relocation type of the value of an expression has nothing in common with its address-numeric type.

In order to avoid ambiguities and mistakes the programmer should use the sticky expression machinery with great care. In general, it seems reasonable to obey the following rules:

i.
Do not use address expressions with absolute values (see the example below, expression no. iii). The authors were not able to find an example which would motivate or provoke such expressions.

ii.
Precede each address symbol occurring in a numeric expression by the monadic operator "#".

iii.
Wherever possible, avoid non-evaluable expressions.

Although it is not likely that any sane Gass programmer would extensively use absolute address expressions, the relocatable numeric expressions cannot be forbidden totally. The crucial idea of our computer is that addresses can be stored and processed as usual data.
However, it is sometimes important to know whether a numeric expression is in fact "numeric", i.e. its value is absolute, or whether it represents a numeric value of an address. In many places the former case is the expected one - we will call it a **constant expression**.

It is now time to follow a number of examples.

Examples

Let us make the following assumptions:

```
muka              is a b-type address symbol,
dobo and bulba    are both c-type address symbols,
outer             is an imported d-type address symbol,
```

and look at the three expressions below:

```
i.     muka + < #dobo - #bulba > + b'1010'
ii.    100 + #dobo * #outer - o'377'
iii.   bulba - < 'c'  !  X'20' > - dobo
```

The first expression reduces to a b-type address value. Gass

evaluates it as follows:

The first operation to be performed is an addition with a subexpression standing for the second operand. This subexpression is evaluated and it produces the absolute difference between "dobo" and "bulba" (note that they have the same relocation types). This absolute value is then added to the address muka retaining its relocation type. So finally is the binary constant b'1010' and the evaluation results in a b-type address value.

Now let us try to follow the evaluation of the second expression.

In the first step the constant 100 is added to the address value dobo and the result is a c-type numeric value. Then that result is to be multiplied by another numeric value, namely "#outer". At this point Gass abandons the evaluation and prepares the expression so that it can be completed by the loader. Nevertheless, it may happen that the evaluation cannot be deferred. In such a case Gass complains about an error and does not bother the loader.

The second expression is a numeric one. However, we can say nothing about the relocation type of its value.

Now the reader can easily guess what is the value of the third expression. It is not hard to deduce that it is the absolute distance between "bulba" and "dobo" decremented by something absolute. In fact, it is not recommended that such expressions be used in practice.

vi. Assigning values to symbols

The general idea of the location field is to provide a place for a symbolic label. Such a label can conceptually represent the target of a jump instruction in a program code or an area where some data is to be kept. In fact, most of the situations when a symbol is put into the location field of a source line result in assigning to the symbol the value of the location counter for the block in use. Although a number of pseudos constitute a great deal of exceptions to this rule, this way of treating the location symbols is thought of as the standard one. In particular, the lines containing symbolic machine instructions conform to the standard.

A convenient (and readable) way to define a symbolic label is to put it into the location field of an empty line (i.e. a line that does not contain anything except the location symbol and possibly some comments).

Some pseudos treat their location fields in a non-stardard way. However, all those which define or reserve data areas are smart enough not to persecute the programmer. Nevertheless, for each pseudo the meaning of its location field will be described separately.

vi.1 .equ - explicit assignment to a symbol

The important pseudo presented in this section is used to assign explicitly a specific value to a symbol. The location symbol must be present and it is assigned a value determined by the arguments of .equ which may take any one of the following shapes:

i. "constant expression"
ii. "evaluable address expression"
iii. "evaluable address expression" ("constant expression")

The first format is used whenever a numeric value is to be assigned. "Constant expression" must reduce to a locally computable number possibly augmented by at most one imported constant.

Formats ii and iii represent address values. The value of the optional "constant expression" is added to the value of "evaluable address expression". In this case the value of "constant expression" cannot depend upon imported constants.

All symbols occurring in the arguments of an .equ **should be defined previously**, i.e. their definitions should lexically precede the line including that .equ.

Examples

Let us assume that the following symbols have already been defined:

```
bytsym              - address symbol, b-type relocation,
abssym              - local absolute constant,
codsym, label       - address symbols, c-type relocation,
datsym              - address symbol, d-type relocation,
impcon              - imported absolute constant,
extern              - external variable, d-type relocation.
```

We will try to analyze a few .equ-assignments.

i.

```
sym1      .equ      bytsym + <label - codsym>
```

"sym1" is defined as an address symbol (b-type relocation). The value of the argument expression reduces to an offset to "bytsym". Let us note that <label - codsym> is absolute.

Anyway the above assignment does not seem too clever. It would be much more in-the-spirit-of-Gass to use the third format of .equ arguments. The assignment would then be:

```
sym1      .equ      bytsym( #label - #codsym )
```

Finally let us note that had we removed the brackets from the original example we would have obtained an erroneous statement. Namely, Gass would not have been able to compute the step "bytsym+#label".

ii.

```
sym2      .equ      abssym & X'aaaa' + impcon
```

Now the location symbol "sym2" is assigned a numeric value. The local symbolic constant "abssym" is logically and'ed with the hexadecimal constant "X'aaaa'" and the resulting number is added to the imported constant "impcon". Thus "sym2" represents a compound numeric value (see iii.3).

iii.

```
sym3      .equ      extern( abssym + 1 )
```

Another compound value. Now it is an offset to the address of an external variable.

iv.

```
sym4      .equ      #extern - #datsym
```

This is an erroneous assignment. Although both "extern" and "datsym" have the same relocation types their difference cannot be computed by Gass. If "datsym" were defined as an offset to "extern" the situation would be quitedefined as an offset to "extern" the situation would be quite different.

Caution

The "numeric-address" expression attribute **is ignored** by the .equ pseudo. In particular, when a numeric relocatable expression is used its "numeric" attribute is stripped off and the expression is assumed to represent an address value. **No numeric-relocatable symbolic values can be defined under Gass**.

There is a very convenient and readable synonym for ".equ", namely the **equal sign "="**. As it turns out, it is much more commonly used than the original mnemonic.

vi.2 Redefinability of symbols

All the symbols defined by .equ (=) can be altered, i.e. they may be redefined by subsequent .equ's or in any other way. Nevertheless, only the .equ-defined symbols have this property. Attempts to redefine other symbols will not succeed and Gass will complain about errors.

vi.3 Changing the location counter

There exists a useful variant of .equ which can be applied when the value of the location counter is to be changed explicitly. Gass recognizes that variant whenever the location field of an .equ assignment contains a period. In such a case the arguments (evaluated as usual) should produce a **relocatable local address value**.

Whenever the new relocation type of the location counter differs from the previous one an implicit .use is forced to switch assembly to the appropriate block.

If the previous block was "byte" Gass performs an **alignment** (see viii) prior to affecting the location counter.

Caution

If the relocation type of the new location counter is "b" then its value **must be even**, i.e. the location counter for the byte block must be adjusted to the full word boundary.

vii. Reservation pseudos

There is a number of situations in which the programmer wants to reserve a specific area of memory. Sometimes it is desirable to preset the reserved locations. The pseudos described in the present section provide the suitable tools.

vii.1 .data

The most frequently applied reservation pseudo is .data which is commonly used whenever a word or sequence of words is to be reserved and preloaded. The format of .data is:

.data "list of elements"

where "list of elements" stands for comma-separated sequence of data items. A location symbol may be present and, if so, it becomes defined as a symbolic address of the origin of the reserved area. The data items describe the values that are to be successively put into the reserved words. Those items may be as follows:

- **numeric constant or any expression (evaluable or non-evaluable)**
- **long constant**

In the first case one word of memory is reserved and filled with the value of the constant or the expression.

In the second case the number of words depends upon the size of the specified constant. For long integer and floating constants this number is 2 and 3 respectively. For a text constant it is equal to the number of characters of the constant. Those characters are converted to the corresponding Ascii codes which go to the right (lower) bytes of the consecutive words.

Caution

Any data item beginning with an apostrophe is always assumed to be a text constant. Therefore, an expression (possibly including character constants) **must not begin with an apostrophe**. This may be achieved by permuting the expression components or by preceding it by a dummy plus sign.

Examples

```
hun     .data     100,200,300,400,500,600,700
```

Seven words are reserved and filled with the binary representations of the specified integers. The symbol "hun" is defined as a symbolic origin of the reserved area.

```
        .data     1.0,'sample text'
```

An area of 14 words is reserved, the first three of them contain the image of floating one, the remaining 11 - Ascii representations of the characters 'sample text'.

.data .

The reserved word contains the value of the location counter for the block in use. This value is equal to the address of that word plus one. **Note that the autoincrementation of the location counter occurrs prior to the reservation**.

```
.data    +'ab'
```

The plus sign preceding the first apostrophe causes the argument of data to be treated as an expression rather than a text constant. One word is reserved which contains the character code of "a" on the left byte and the code of "b" on the right.

```
.data    'ab'+1
```

Illegal data - a text constant cannot be followed by a plus sign

vii.2 .text

The .data pseudo does not seem too reasonable when the reserved area has to be filled with a text. The manner in which Gass stores text constants of .data is both memory-expensive (one character per word) and not susceptible to the byte instructions of the computer.

The text pseudo is used to fill a specified area with text in such a way that subsequent characters are placed in subsequent bytes rather than words. The acceptable formats of the .text pseudo are:

```
.text    "text constant"
.text    "constant expression","text constant"
.text    "constant expression","adjustment","text constant"
```

The first format reserves [(n+1)/2] words of memory, where n equals to the number of characters of the text constant. The Ascii representations of the characters of the string enclosed in apostrophes are assigned to consecutive bytes of the reserved area. If the number of those characters is odd then the right (lower) byte of the last reserved word is filled with zero.

The two remaining formats are used to reserve a predefined number of words and fill them (perhaps partially) with the given text. The value of "constant expression" determines the number of words to be reserved. These words are filled with the specified text which may be left or right justified. The default adjustment is "left" and it is assumed wherever the second format is used. The optional "adjustment" may be one of the letters "l" or "r". The first case does not differ from the default ("adjustment" absent), the second - forces right adjustment.

The number of words indicated by the value of "constant expression" must be large enough to contain the specified text. In the opposite case an error is flagged and the area is not reserved.

A location symbol may be present and, if so, it is treated in the usual, standard way.

Examples

```
header    .text    'page number nnnn'
```

Eight words are reserved. Subsequent bytes contain the character codes of the specified text.

```
                    .text     'abc'
```

The number of characters is odd and the second reserved word is right-filled with zero.

```
                    .text     10,'left justified'
```

An area of ten words is reserved. The first seven of them contain the image of the text 'left justified', the remaining three are filled with zeros.

```
                    .text     10,r,'right justified'
```

The first five bytes of the reserved area are filled with zeros. The remaining fifteen bytes contain the codes of the specified characters.

vii.3 .res

In many cases when an array is reserved the problem of presetting its items is a trivial one. Most commonly, the array does not have to be preset at all; sometimes uniform values have to be preassigned to all its items.

The recommended pseudo for such an occasion is .res which can occur in one of the following two formats:

```
          .res      "constant expression"
          .res      "constant expression","constant expression"
```

The first "constant expression" (the only "constant expression" for the first format) determines the number of words to be reserved. For the second format, it is guaranteed that the reserved words will be filled with the specified value (the value of the second "constant expression"). **Nothing can be presumed about the contents of those words when the first format is used**.

As usual, an optional location symbol may be present to tag the beginning of the reserved area.

Examples

```
                    table     .res      100+b'10'
```

An area of 102 words is reserved which initially contains garbage.

```
                     array     .res      1000,0
```

An array of 1000 words is defined. All its elements are preset to zero.

viii. Specific pseudos of the byte block

The properties of the byte block make it essentially different from the two remaining ones. The reason why the byte block is such a peculiar feature of Gass is mainly a consequence of the byte address format. This format differs from that of the word address and may be expressed by the following statement:

ba = wa*2 + bn

where:

ba - address of a byte,
wa - address of the word containing this byte,
bn - byte number (0 = upper byte, 1 = lower byte)

During assembly the byte block may be switched between two complementary states, namely the byte block may be:

aligned - when the value of its location counter is even, i.e. the location counter points to a full word

or

unaligned - when the location counter points to the right byte of a word, i.e. the value of the location counter is odd.

If the byte block is aligned then all the reservation pseudos (and the symbolic machine instructions) described elsewhere in this manual can freely apply to that block. Let us note that they will never turn the byte block to the unaligned state as the number of bytes they reserve is always even. Nevertheless, there are some pseudos, applicable only to the byte block, which may occasionally reserve an odd number of bytes. Because the unaligned byte block is a cripple which cannot be processed by the standard means of Gass, a special pseudo has been provided to force the alignment explicitly. This pseudo is simply:

.align

If the byte block is already aligned then the .align pseudo is a do-nothing one. Otherwise an extra zero byte is produced in order to adjust the location counter to the next full word boundary.
The location field is processed by .align in the standard way; any symbol occurring there is set to the value of the byte-block location counter as it was before the alignment. .align expects no arguments and, if present, they cause errors.

viii.1 .byte

The .byte pseudo applies exclusively to the byte block and is used to reserve and preset a number of bytes. This number may be to
reserve and preset a number of bytes. This number may be odd, i.e. a partial word may be reserved. The format of the .byte pseudo is:

.byte "list of elements"

"List of elements" is a sequence of data items separated by commas. Each item may be either a constant expression or a text constant. The value of a constant expression is truncated to the lower 8 bits and the

resulting pattern is used to fill a subsequent byte of the reserved area. The characters of a text constant go into consecutive bytes until the constant is exhausted. No extra byte is generated - odd number of characters results in odd number of reserved bytes.

Examples

```
somebytes     .byte     1,2,3,4,5,6,7,8,9,10
```

Ten consecutive bytes are reserved and filled with subsequent integer values.

```
              .byte      'a few characters',X'12f1'
```

Seventeen bytes are reserved: the first sixteen of them contain the charcter codes of 'a few characters', the remaining one - hexadecimal value X'f1'.

viii.2 .resb

This pseudo is an analogon of .res (see vii.3) tailored to reserve bytes rather than words. The arguments are exactly as those of .res: the first describes the number of bytes to be reserved, the second - an optional value to be preassigned to the reserved bytes.

Examples

```
                bytearray    .resb     211
```

An array of 211 bytes is reserved which initially contains garbage.

```
                      .resb     2*<10+20>,'0'
```

Sixty bytes are reserved and preset to the character code of "0" (X'30').

ix. Machine instruction format

The most common contents of operation fields are symbolic machine instructions. In fact, all the remaining features appearing there are merely auxiliary means only used to facilitate the developement of an assembler program. A detailed description of the machine instruction set is contained in the second part of this manual. In the present section the reader will find some details concerning the instruction format and the general semantics of the arguments.

An input line containing a symbolic instruction does not differ too much from any other input line. A location symbol may be present and, if so, it is defined as a symbolic address of that instruction. The shape of arguments depends mostly on the instruction in question - various instructions expect various formats of arguments. The most trivial instructions, with respect to the complexity of their arguments, are those that do not require arguments at all. However, for the purpose of this section, the most interesting will be those instructions which do not fall into that trivial category.

ix.1 Register designators

It is not amazing that the argument syntax commonly admits for register designators. The eight general purpose registers of the Mera-400 processor are represented by the following symbols:

R0 R1 R2 R3 R4 R5 R6 R7

The case of the letter "R" is ignored - in particular, the lower case "r" can freely be used with the same results.

It should be emphasized that the symbolic **register designators are not reserved identifiers**, i.e. they can be used as names of user's private objects. In some contexts where it may cause ambiguities the interpretation as registers designators takes precedence. Nevertheless, there exists an assembly option (see Appendix B) which, if properly set, changes the register designators by substituting the **dollar ("$")** for the letter "r". This possibility may be particulary useful when Gass is used to postassemble the output of a compiler which now does not need to check the names against possible conflicts.

ix.2 First argument

An instruction may produce some result which is returned into one (or more) of the universal registers of the processor. Sometimes, particularly for the instructions that return long values, the output registers are determined and fixed - as examples we can take the floating instructions which always operate on the triplet r1,r2,r3. However, in many cases the result register must be indicated explicitly by the programmer - if so, it is called the first argument of an instruction. The argument field of an input line containing such an instruction should always take one of the following forms:

"register designator","something else" or sometimes simply:
"register designator" (and nothing else)

for instance:

```
set    r3,1981          or
inc    r0
```

Obviously there is nothing strange in the fact that there are some instructions which expect only the first argument. It is not hard to guess that the inc instruction adds one to the contents of the specified register.

The property of "possessing a first argument" should be thought of as a purely syntactic one. In fact, even a jump instruction (which does not admit for a first argument) produces some result and locates it in some register. Namely, the result is the address to where the jump is being performed and the register is obviously the program counter.

ix.3 Second argument

In general, an instruction may demand two operands - the integer 16-bit two-complement addition is an example. It seems obvious that apart from the first argument, it should require another one - namely, an entity to be added to the contents of the first-argument register. Sometimes it happens that the second argument of an instruction occurs while the first is absent. For instance, floating addition always assumes that the first argument occuppies the triplet (r1,r2,r3). Nonetheless, the second argument of this addition should be written explicitly. The shape of the second argument reflects the hardware format of an instruction and therefore it is the most important criterion upon which the instructions are classified. The binary formats of instructions are presented in the second part of this manual. Should they interest the reader just now, she is advised to look a few pages ahead.

The most general and specific format of the second argument is the so called standard format - it will be the first subject of the present section.

Standard second argument

The primary value of the standard second argument is always a 16-bit pattern representing one of the following things:

i. a numeric or address expression,
ii. the contents of a specified register,
iii. anything above augmented by the contents of a register.

The facility expressed by iii is called **indexing**, the register used to modify the primary value is called the **index register**.

The registers used to make up the primary value of a standard second argument may be any universal registers except r0 - this particular register can only be used as a first argument.

The primary value may be related to the location of the argument proper in three ways, each stepping one level into indirection. Namely,

the argument may be:

i. the primary value itself,
ii. determined by the location indicated by the primary value,
iii. determined by the location indicated by the contents of the location indicated by the primary value.

However, any particular standard instruction only admits for two adjacent levels of indirection. Depending on what those levels are the standard instructions are divided into two classes. The first class includes **single indirect instructions** - each of them accepts the second arguments of the levels i and ii, the members of the second class (**double indirect instructions**) admit for the levels ii and iii.

Presented below is the syntax of the standard second argument. The following abbreviations are used to clarify the picture:

c - constant expression
n - numeric expression
a - address expression
r - register designator (r1-r7)

Direct format

n r r+n r+r

Examples

```
set    r1,2000
ior    r0,r3+label-1
add    r7,r7+r7
```

First indirect format

a a(c) a(r) a(r+c) (n) (r) (r+n) (r+r)

Examples

```
set    r1,label              ; (label  is an address symbol)
ldf       float(r7+3)        ; (float  is an address symbol)
ldf       float+3(r7)        ; clumsy equivalent of the above
buc       (1)
ldl       (r7+r6)
mul       (#multiplier)
```

Second indirect format

(a) (a(c)) (a(r)) (a(r+c)) ((n)) ((r)) ((r+n)) ((r+r))

Examples

```
        ldl       (double)           ; (double is an address symbol)
        buc       ((<#symbol-100>/2))
        ldf       (float(r4+3))      ; (float  is an address symbol)
        iorm.  r5,(tag(r7+100))      ; (tag    is an address symbol)
```

The admitted indirection levels of the standard instructions are listed in the second part of this manual. Unfortunately, sometimes they do not follow intuition. A particularly unclever example is the integer multiply instruction which is not able to multiply the first argument by an immediate constant, i.e. the multiplier must reside in memory anyway.

The binary image of a standard-type instruction can occupy one or two machine words. The shorter image results whenever the second argument of an instruction does not contain any expressions. If this is not the case the value (or sum of values) of the expression(s) is put into the second word.

Short-relative argument

Sometimes the programmer desires to access some data or jump to a program location which is not too distant from the place of the reference. In many cases such a reference can be performed with a short (one-word) instruction, provided that the distance in question does not exceed 63 words. Some of the machine instructions expect a special type of the second argument - its value describes the so called displacement, i.e. the distance between the current program location and the location to where the reference is to be performed. The restrictions imposed on that distance are conequences of the size of the binary field used to include it - this size is 6 bits for the absolute value plus one extra bit for the sign.

There are two formats of the short-relative argument, the simplest but less convenient and less frequently used is:

"constant expression"

The absolute numeric value of the expression is viewed as an absolute distance between the current code location and the object to be accessed.

Examples

```
 l.rg   r1,3               ; r1:=the contents of the fourth cell down
 b.uc      -50             ; jump 50 words back
```

Let us note that **the current cell means in fact the first cell following the image of the current instruction**, for example:

```
                            b.uc      -1
```

is an instruction which unconditionally jumps to itself.

Using absolute displacements does not seem to be a wise methodology, at least as far as the distance is not trivial and obviously seen. It is much better to use constructions of the following form:

```
label    inc    r1
    ..................
         b.eq      #label-#.
```

which express the distance in terms of symbolic labels. Anyway, the most advised format of the short-relative argument is the second one, namely:

"evaluable address expression" or
"evaluable address expression"("constant expression")

The value of the expression (sum of values for the second format) must reduce to a relocatable address value of the relocation type which corresponds to the type of the block in use. The absolute difference between the value of the expression and the value of the location counter for the current block may not exceed 63.

Examples

```
    ......................
label    l.byte  r4,arry(r3)
         cmps    r4,'e'
    ......................
         b.uc       label         ; short jump to "label"
    ......................


forever  b.uc       forever       ; infinite loop


    ......................
imdata   .data       0,1,2,3,4,5
*
*
    ......................
         l.rg    r7,imdata(3)     ; fourth item of 'imdata'


cells    .res       65            ; reserve 65 memory cells
    ......................
         s.rg    r4,cells(1)      ; error - distance too long
    ......................
```

Short-numeric argument

Using standard instructions to perform simple arithmetic operations which involve small immediate integer operands seems to be a waste of memory space; let us remind that the standard second argument including any constant requires one extra word to locate it. On the other hand, plenty of computational arithmetic makes use of some short typical numbers as 0, +1, -1 and so on. Therefore, there exist a number of machine instructions which facilitate such operations - the type of the second argument they expect is called the short-numeric one. In fact,

this type is very simple - any valid constant expression can stand for the short-numeric argument, provided that its absolute value does not exceed 63.

Examples

```
adds   r1,4                  ; r1:=r1+4
lds    r7,2*<7+b'11'>        ; r7:=20
```

All the places where standard instructions can be substituted by their shortened correspondents are flagged by Gass on the listing (see xiv). For all those who do not consider counting to 63 to be an especially interesting job Gass offers an interesting assembly option (cf Appendix B). If this option is turned on (and the so called "optimizing run" is selected) then Gass produces a copy of the source program text instead of the binaries. In this copy those instructions which Gass would flag as susceptible for abbreviation are exchanged by their abbreviated versions. The total number of words recovered by this action is printed. If this number is substantially greater then zero a next iteration may be required, as some distances in the program became reduced, what could result in further instructions susceptible for abbreviation.

Byte argument

The last small class of instructions contains those which expect the second argument to represent a value which can fit into one byte. A constant expression which can appear there must possess a nonnegative value not exceeding 255.

Examples

```
if.sbs    255       ; execute next instruction if scratch byte is nonzero
ptr       70        ; request program trap #70
```

x. Macroassembly

Gass, as well as other reasonable assemblers, is equipped with a pertinent macro-machinery. This apparatus allows for defining as macros some pieces of the text to be assembled and then to reference them by names in all places they have to be inserted. Moreover, there is a possibility to parametrize those pieces, so that their various extensions may differ appropriately.

x.1 Defining macros

In order to define a macro the programmer should put the desired sequence between the following pseudos:

```
.macro
--
--
.emc
```

The location field of the .macro pseudo should be nonempty - it should contain the symbolic name to identify the macro. All the macro names have to be unique and they cannot be re-used to identify other objects.

The sequence to be defined as a macro cannot contain another macro definition, i.e. macros cannot be defined recursively. Nevertheless, the recursive macro reference is totally allowed and turns out to be a common practice.

A reference to a macro must be lexically preceded by a definition of this macro. Macro-definitions can appear anywhere in the source text except for the macro-definitions themselves.

Examples

The following macro generates a sequence to skip one instruction if r1 does not contain the character code of a blank:

```
if.blank    .macro
             cmps    r1,' '
             if.eq
            .emc
```

Now, some sample code using the above macro:

```
          ......................
nextchar   l.byte  r1,bytes(r2)      ; fetch next byte
           inc     r2                 ; update counter
           if.blank                   ; ignore blanks
           b.uc       nextchar
          ......................
```

Another example of a simple macro is the following sequence to convert an integer number from r1 to the floating format into r1,r2,r3:

```
float       .macro
             clr     r2          ; prepare mantissa
             lds     r3,15       ; and exponent
             norm                ; and normalize
            .emc
```

Anyway, the above two macros do not seem too smart. It is much wiser to parametrize the argument registers.

x.2 Formal parameters

Within the text of a macro-definition there may occur sequences of the following form:

?"decimal digit"

Each such a sequence represents a reference to the formal argument number i, where i is the value of "decimal digit". The digit 0 (zero) is reserved to represent the argument count, i.e. the sequence ?0 expands to a digit which denotes the number of the arguments upon the current macro-reference. Thus, it is not hard to deduce that **the maximum number of the arguments of a macro is nine**.

The idea of passing the actual parameters upon a reference of a macro-prototype is very simple, but the solution turned out to be powerful enough. The point is that the parameters are viewed as texts and they are simply substituted (rewritten) for their formal correspondents. This substitution is never validated nor is any interpretation of the parameters performed until the moment when the expanded macro text is assembled.

x.3 Actual parameters

The actual parameters of a macro-reference are simply pieces of text separated by commas or blanks. Two or more consecutive blanks stand for one, any blanks following a comma delimiter are ignored. Two consecutive commas optionally separated by blanks denote an empty parameter. All the empty parameters are null texts - references to them are legal and produce obvious results.

Sometimes it may happen that a blank or comma is to be included within a parameter text. Two ways of solving such problems can be advised. The first one, and less pleasant, is to precede the goodie in trouble by a backslash. Although the unfortunate backslash is passed together with its follower this method happens to work quite well. Nonetheless, the second method is much more general and wise. Namely, whenever an actual parameter begins with an apostrophe (='=) Gass assumes that it continues over until the second matching apostrophe is encountered. Then both the apostrophes are removed and the entire text between them is passed as the actual parameter. The only disadvantage of this gadget is in situations when an apostrophe is to be included within the parameter text. Fortunately there is a solution to this trouble, namely, such an apostrophe has to be typed twice.

Examples

Let us redefine the if.blank macro of the previous example, now the register to be examined will be parametrized.

```
if.blank    .macro
             cmps    ?1,' '
             if.eq
            .emc
```

Now, when referring to the above macro, the programmer should pass to it a pertinent register descriptor, for instance, the reference may be:

```
if.blank  r4
```

For another example let us consider a macro to define a text and to force an extra zero word to terminate it. Here is one way of defining this macro:

```
.display    .macro
            .text       '?1'
            .data       0
            .emc
```

Now a few sample references:

```
   ........................
            .display    atextwithnoblanks
   ........................
label001    .display    'here blanks can freely occur'
   ........................
            .display          ; reserve one zero word
   ........................
```

Of course, the mostly recommended format of the parameter for the .display macro is that used in the second reference (it reflects the text nature of the parameter).

The .display macro is a pseudo-like one, i.e. it is defined to simulate a pseudo rather than a machine instruction. Although Gass is not bothered by this kind of matters it seems to be a good habit to begin the names of such macros by a period.

x.4 Conditional macro expansion

There exist some pseudos which, when used within a macro, can fork the way the macro is being expanded: depending on certain conditions examined by those pseudos the macro can expand in various ways. In particular, loops can be organized and non-trivial recursive references to the macro itself can be performed.

Although the basic set of means to realize the conditional expansion of macros is amazingly simple, it turns out to be an extremely powerful tool, particulary, when applied together with the micro-gadget (see xi).

Numeric meta-labels

Any location field containing an unsigned natural number is ignored by Gass in the instant the line is assembled. The numbers occurring in location fields may serve as a kind of comments or as the meta-labels addressable by the tools of the conditional assembly.

A tailored subset of the conditional-assembly machinery is accessible and applicable also outside macros (see xii). Of course the assembly loops are not available there.

The following, simplest conditional-assembly pseudo is used wherever an unconditional meta-jump within a macro is to be performed:

.skip "natural number" (optionally preceded by a sign)

"Natural number" represents the meta-label to where the macro expansion is to be switched. The label must be defined somewhere within the macro, otherwise an error is flagged and the pseudo is ignored.

The target line of the .skip macro is looked for starting from the next line down (wrapping around when the end of macro is reached). The first line which contains the desired label is assumed to be the one in question; thus the meta-labels do not have to be unique.

In order to speed up the process of macro-expansion the programmer may change the standard way of searching for the target label. Namely, "natural number" can optionally be preceded by a sign and, if so, the following results will be obtained:

+

The standard (default) way of searching is selected.

-

The target line is looked for starting from the beginning of the macro and going to the end.

The following conditional pseudos are used to perform appropriate meta-jumps upon the suitable conditions - the way of performing these jumps is that described above:

.ifzr "constant expression","possibly signed natural number"

The meta-jump is performed if the value of "constant expression" is zero.

.ifpl "constant expression","possibly signed natural number"

The meta-jump is performed if the value of "constant expression" is positive.

.ifng "constant expression","possibly signed natural number"

The meta-jump is performed if the value of "constant expression" is negative.

.ifdf "nonzero decimal digit","possibly signed natural number"

The meta-jump is performed if the parameter indicated by the digit is present.

.ifun "nonzero decimal digit","possibly signed natural number"

The meta-jump is performed if the parameter indicated by the digit is absent.

.ifdf "symbolic name","possibly signed natural number"

The-meta jump is performed if the symbolic name is defined.

.ifun "symbolic name","possibly signed natural number"

The meta-jump is performed if the symbolic name is undefined.

.ifrg "text","possibly signed natural number"

The meta-jump is performed if the specified text is a valid register designator.

.ifnr "text","possibly signed natural number"

The meta-jump is performed if the specified text is not a valid register designator.

Note

Whenever an error is detected during evaluation of the argument expression in an arithmetic conditional pseudo (.ifzr, .ifpl, .ifng) then no standard error action is performed, but the pseudo behaves as if the specified condition were **not satisfied**.

Let us follow some examples:

Examples

The following macro assigns the factorial of its second parameter to the first parameter:

```
.factorial   .macro
?1      =           1              ; initialize resulting symbol
temp::: =           ?2
1      .ifzr        temp:::,2      ; terminate if end of loop
?1      =           ?1 * temp:::   ; multiply result
temp::: =           temp::: - 1    ; and decrement multiplier
       .skip        1              ; loop till multiplier reaches 0
2      .emc
```

Another macro presented below reserves an area of n words (n is the only parameter) and fills the reserved area with the subsequent integer numbers:

```
.area       .macro
temp:::      =        0              ; starting filler
1           .ifzr     temp:::-?1,2   ; check for the exit from loop
temp:::      =        temp::: + 1    ; update filler
            .data     temp:::        ; reserve next word
            .skip     1              ; loop for more words
2           .emc
```

Sample references to the above two macros may look as follows:

```
.........................
    .factorial fact,4         ; fact   =   4!
.........................
    .area      1000           ; 1000 words especially filled
.........................
    .factorial n,#lab1-#lab2  ; n   =   (lab1-lab2)!
.........................
    .area      3+<3*9>
.........................
```

xi. Micros

Before Gass starts to assemble the current line of its input it first looks for all the sequences of the following form:

"double quote""symbolic name"

Wherever such a sequence (micro-reference) is found Gass tries to replace it by the decimal positive representation of the numeric value assigned to the symbolic name. When this name is undefined or its value is not numeric an error is assumed and the line is no longer interpreted. The numeric values of micro-symbols are treated as positive integers with the leftmost bit interpreted as a significant binary position rather than a sign. The leading zeros of the decimals to which the micros are converted are stripped off. The only exception is the zero value which is converted to the one digit "0".

Examples

Let us assume that somewhere up in our program the following equalities have been defined:

```
sym1      =      0
sym2      =      1
year      =      1987
negative  =     -1
```

Then the micro-references below will be converted as follows:

```
.text 'the digit is "sym1'   ===>   .text 'the digit is 0'
b.uc  label"sym2             ===>            b.uc  label1
we have "year                ===>            we have 1987
positive -1 is "negative     ===>    positive -1 is 65535
```

Micro-references can be nested recursively and intermixed with references to macro parameters. If so, the interpretation goes **from right to left** as reflected by the following examples:

Let us borrow the symbol definitions from the previous example and assume that the following micro references are performed within some macro; the macro has been called with two parameters: "year" and "5" respectively. The conversions will proceed as follows:

```
.data "sym1,"sym"sym2        ===>              .data 0,0
jun 12, ?"sym2               ===>           jun 12, 1987
no. of params is ?"sym"sym2  ===>      no. of params is 2
```

The applications of micros are many and they cannot be described completely in this chapter. The most powerful application seems to be the automated creation of distinct symbols, particulary useful within macros. Many other clever examples of effective micro use go together with sophisticated macros. For instance, let us consider the one below:

Example

The presented macro generates an unwound code for computing the value of a floating polynomial. The input argument is assumed to reside in r1,r2,r3, the coefficients are passed as subsequent parameters upon the macro-reference.

```
flpoly    .macro
          .ifzr      ?0,1            ; if nothing to do
*
* Reserve area for coefficients and argument and plug them there
*
          .use       data            ; assume data block
          .ifdf      c::,4           ; if not first reference
*
*      The symbol c:: will  be  used to count references to this macro.
* By means of this symbol we will be able to define uniformly the
* labels of the data area generated by the subsequent references.
*
c::        =         0               ; first reference - define c::
          .skip      5
4                                    ; not first reference - update c::
c::        =         c:: + 1
5
*
temp2::    =         1               ; start from the first parameter
t::"c::   .res       3               ; area for the argument
*
* Meta-loop thru the parameters
*
2         .data      ?"temp2::       ; store next parameter
temp2::    =         temp2::+1       ; update parameters counter
          .ifpl      temp2::+1-?0,2  ; if more parameters
*
* The parameter values have been sent to memory - now generate code
*
          .use       code
           strf      t::"c::         ; store argument
           ldf       t::"c::+3       ; load n-th coefficient
temp2::    =         1               ; initialize coefficient number
*
*   Meta-loop thru coefficients
*
3         .ifzr      temp2::-?0,1    ; if all coefficients done
           mulf      t::"c::         ; multiply by the argument
temp2::    =         temp2::+1       ; increment coefficient counter
           addf      t::"c::+3*temp2:: ; add the following coefficient
          .skip      3               ; loop for more coefficients
*
1         .emc
```

xii. Parametrized assembly

When preparing a release version of a software product it often becomes neccesary to rewrite some pieces of code in a few alternative versions in order to meet the anticipated demands of various users or to tailor the product to the specific hardware configuration of the target computer. It would be nice if assembly of particular pieces were turned on or off by some simple means, for instance, by defining some symbolic names or (and) by setting them to appropriate values. Gass accomplishes these requirements by the tools of conditional assembly described in x.3 which are accessible also outside macros. The numeric meta-labels described there can also appear outside macros and their properties are retained. The only difference is that no backward meta-jumps can be performed, i.e. the meta-loops are exclusive gadgets of macros. Moreover, the variant of .ifdf (.ifun) allowing the first argument to be a digit is illegal. Let us remind that this variant is used to ask about the presence of an actual macro parameter.

Example

Let us look at the following fragment:

```
          ..............................
              .ifzr     version-1,1      ; ignore if version 1
  *
  *   Conversion table for version two
  *
  convtable   .byte     'a','b','c','d','e','0','1','2','3','4'
              .skip     2
  *
  *   Conversion table for version one
  *
  1           .byte     'f','g','h','i','j','5','6','7','8','9'
  2
          ..............................
```

Depending on the value of the symbol "version" the appropriate byte table is selected.

xiii. External assembly

The units of a large program frequently require a great deal of common text to be inserted into each of them. Although the problem of physical insertion of such a text can easily be alleviated by a powerful text editor, it is sometimes better to avoid duplication, particulary when a lot of disk space can be saved this way. Fortunately, Gass is equipped with a suitable tool which allows it to switch assembly to an external file and, after exhausting its contents, to return to the original source. Such an action is attempted whenever the following pseudo is encountered:

.file '"file name"'

The file "file name" is rewound and its contents are assembled as well as they would have occurred in the original source. After the file is exhausted assembly returns to the first line following the .file pseudo. External assemblies can be nested, i.e. an external text may contain another .file pseudo.

Restrictions

The .unit and .end pseudos are inhibited in an external text, i.e. an external text cannot be continued over more than one unit. Finally, there is one more restriction, namely, for a technical reason the .file pseudo is inaccessible from macros.

xiv. Listing

Whenever called with a suitable option (see Appendix B) Gass produces an assembly listing which carries out some information about the program lines being assembled. Two versions of listing are accessible - the first one (partial listing) produces at most one listing line for each source line. Macro extensions are not listed, nor are external texts and listing control pseudos. The second (full) version of listing produces exactly one listing line for each line interpreted by Gass regardless of what the source of that line is.

Each listing page begins with a header which includes the name and release version number of the assembler as well as the date, time and page number. A typical listing line begins with a hexadecimal number representing the current location followed by a letter denoting the current block. If the line contains some pseudo or machine instruction which fills a number of storage cells then the images of two initial cells of that number are printed. Also the .equ (=) pseudo produces a printout of the value assigned to its location symbol. Whenever a line is an erroneous one a corresponding error flag is included immediately following the image of the block location counter. This flag has the following form:

@"character"

where "character" denotes the error type (refer to Appendix a for description of error types).

The listing line location reserved for error flags serves yet another purpose. Namely, whenever Gass detects that the current instruction **can be abbreviated**, i.e. it can be replaced by its corresponding displacement instruction (see ix.3), the sequence "**>>**" (double greater) appears in that location.

When the listing of a unit is completed Gass follows it by a printout of the symbol table. All the symbols defined or referenced in the unit are sorted by names and these names are printed out followed by values and descriptors. Each such descriptor may be composed of the following letters:

A,B,C,D	-	representing the relocation type of the symbol,
M	-	for the name of a macro,
E	-	if the symbol is exported,
V	-	for an external variable,
I	-	for an imported symbol,
T	-	for a transfer address.

xiv.1 Listing control pseudos

The three pseudos presented here are used exclusively to control the listing - they do not influence assembly and do not affect the resulting program code.

.eject

Gass changes listing page before printing the next listing line.

.head "text constant"

The argument text will be included in the header of each succeeding page. A new page is forced as for the .eject pseudo.

.list "listing version"

"Listing version" may be one of the following keywords:

ON	-	to select the full listing version,
PARTIAL	-	to select the abbreviated listing version,
OFF	-	to switch the listing off.

The indicated listing version is selected for the current unit until another .list is encountered. Anyway, the .list pseudo is only active when the (partial) listing version has been selected upon calling Gass.

The location fields of the listing control pseudos are ignored and may be used to include comments or numeric meta-labels.

xv. Auxiliary pseudos

There are two special pseudos of Gass which should be put into a separate category, they are:

.rem "text constant"

The argument text of the above pseudo is put into the binary comment table of the current unit.

.error "text constant"

This pseudo forces unconditionally an error flag. "Text constant" is optional and, if present, the text is printed onto the operator console. The .error pseudo is recommended for use whenever some unwanted results may arise in conditional assembly.

The location fields of the auxiliary pseudos are ignored.

Appendix A

Errors

Some of the errors which can occur during assembly make the assembler unable to continue. In the case of such an error the following actions are undertaken:

1. the last line being assembled is printed out,
2. the text "X-type abort" (X denotes the error type) is printed on the user's standard output,
3. assembly is aborted.

The following error types are as serious as to abort assembly:

T - symbol table overflow,
Q - macro table overflow,
V - stack overflow while expanding macro.

The remaining errors do not affect assembly except for the erroneous lines. If any error is detected in a line being assembled then the remainder of the line is ignored and assembly re-starts at the beginning of the next line. This means that only one error per line can be detected at a time. The errors are flagged by the corresponding error-type characters which appear on the listing and are preceded by "@". Below is the list of all possible error types:

L - location field illegal or unrecognizable,
O - operation field illegal or unrecognizable,
I - bad item of expression,
U - undefined symbol,
G - illegal symbol attribute,
V - non-serious stack overflow,
C - constant expression expected,
E - bad expression syntax,
S - illegal second argument of a machine instruction,
R - value out of range,
B - evaluable expression expected,
N - symbol missing,
P - too many arguments,
H - pseudo is illegal in macro or external file,
D - double symbol definition,
T - bad text constant,
A - illegal arguments of a pseudo,
M - illegal name of a macro,
F - numeric constant overflow,
Y - micro reference error,
Z - macro parameter error,
W - input line too long,
+ - too many imports or external variables,
K - meta-label not found,
- - non-aligned byte block.

Appendix B
Using Gass under CROOK-5 operating system

Calling Gass

To invoke the assembler the following macro is commonly used:

```
gass input.file output.file [options]
```

The first two parameters are file names, one for the input (source) file, the other for the output (usually binary) file. The following options can be used:

?REG - changes the register mnemonics to $0, $1, ..., $7;
?OPT - selects the optimizing run (see ix.3), the optimized source text appears on the specified output file;
?SIZ - declares the requested symbol table size - the option should be followed by an integer parameter, specifying the table length in units of 100 entries. The standard value is 15 (1500 entries), the maximum value is 30.

The following options listed require a file name following their appearance.

?NOL - (no listing) only errorneus lines will be directed to the specified file;
?MAP - only symbol table will be directed to the specified file;
?LIS - the source text directives controlled listing will be placed in the specified file;
?FUL - full listing is forced.

All options may, but don't need to start with the question mark and are recognized basing on their first letter, regardless of its case. If no listing control option is specified, the result is as if '?NOL MON' were typed.

Examples

```
gass source1 source2 opt ?lis abc
gass program modul
gass driver mmm reg ?S 30
gass game ggg ?nol temp
```